**COGNIZANT JAVA FSE PROGRAM**

**WEEK 1:ALGORITHMS\_DATA\_STRUCTURES**

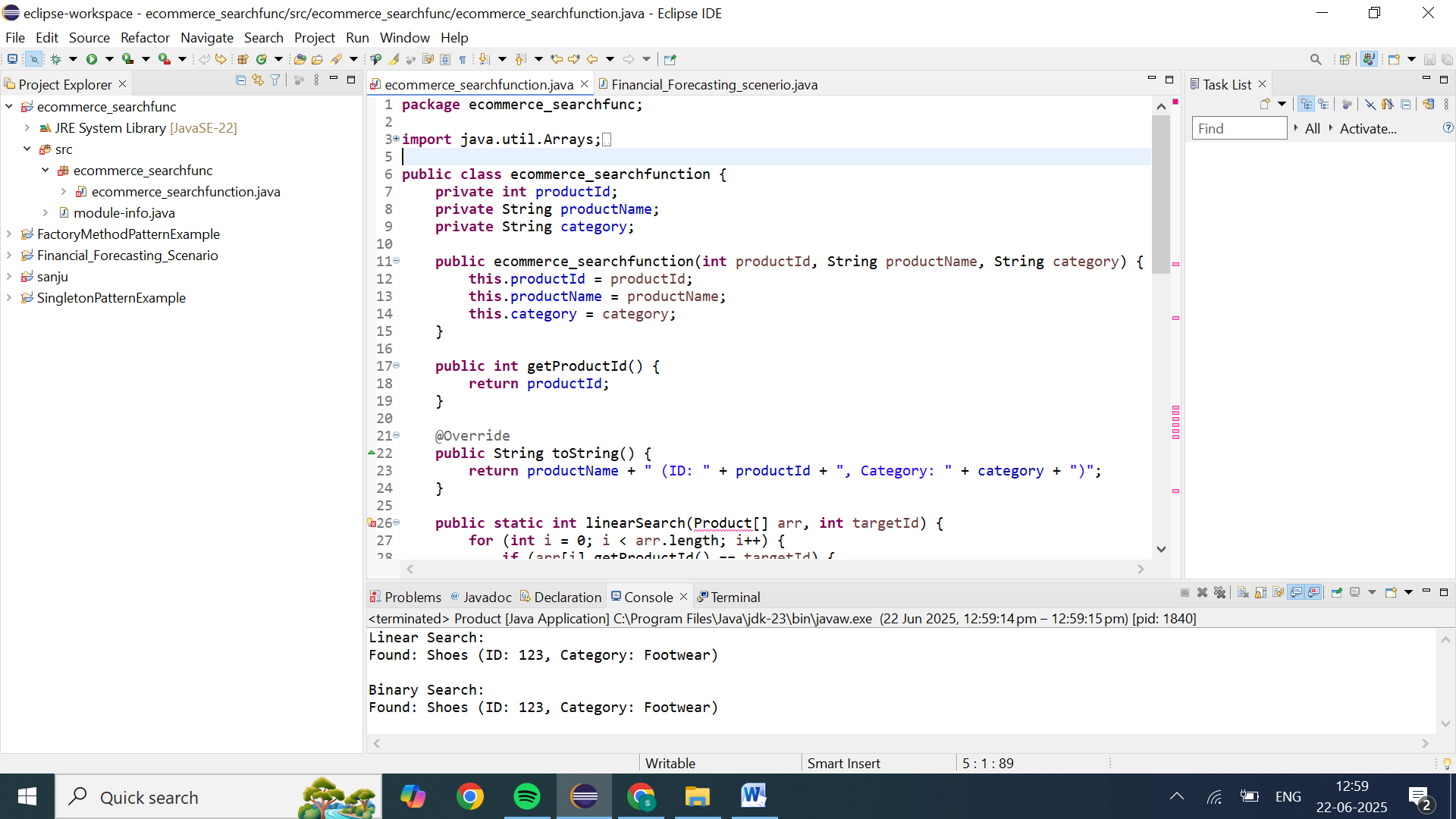
**1: E-commerce Platform Search Function**

**Scenario:**

You are working on the search functionality of an e-commerce platform. The search needs to be optimized for fast performance.

**Steps:**

1. **Understand Asymptotic Notation:**
   * Explain Big O notation and how it helps in analyzing algorithms.
   * Describe the best, average, and worst-case scenarios for search operations.
2. **Setup:**
   * Create a class **Product** with attributes for searching, such as **productId, productName**, and **category**.
3. **Implementation:**
   * Implement linear search and binary search algorithms.
   * Store products in an array for linear search and a sorted array for binary search.
4. **Analysis:**
   * Compare the time complexity of linear and binary search algorithms.
   * Discuss which algorithm is more suitable for your platform and why.



**package** ecommerce\_searchfunc;

**import** java.util.Arrays;

**import** java.util.Comparator;

**public** **class** ecommerce\_searchfunction {

**private** **int** productId;

**private** String productName;

**private** String category;

**public** ecommerce\_searchfunction(**int** productId, String productName, String category) {

**this**.productId = productId;

**this**.productName = productName;

**this**.category = category;

}

**public** **int** getProductId() {

**return** productId;

}

@Override

**public** String toString() {

**return** productName + " (ID: " + productId + ", Category: " + category + ")";

}

**public** **static** **int** linearSearch(Product[] arr, **int** targetId) {

**for** (**int** i = 0; i < arr.length; i++) {

**if** (arr[i].getProductId() == targetId) {

**return** i;

}

}

**return** -1;

}

**public** **static** **int** binarySearch(Product[] arr, **int** targetId) {

**int** left = 0, right = arr.length - 1;

**while** (left <= right) {

// prevent overflow :contentReference[oaicite:1]{index=1}

**int** mid = left + (right - left) / 2;

**int** midId = arr[mid].getProductId();

**if** (midId == targetId) **return** mid;

**if** (midId < targetId) left = mid + 1;

**else** right = mid - 1;

}

**return** -1;

}

**public** **static** **void** main(String[] args) {

Product[] products = {

**new** Product(102, "Phone", "Electronics"),

**new** Product(205, "Shirt", "Apparel"),

**new** Product(315, "Laptop", "Electronics"),

**new** Product(123, "Shoes", "Footwear"),

**new** Product(412, "Book", "Stationery")

};

**int** targetId = 123;

System.***out***.println("Linear Search:");

**int** idx1 = *linearSearch*(products, targetId);

System.***out***.println(idx1 >= 0 ? "Found: " + products[idx1]

: "Product not found.");

Arrays.*sort*(products, Comparator.*comparingInt*(Product::getProductId));

System.***out***.println("\nBinary Search:");

**int** idx2 = *binarySearch*(products, targetId);

System.***out***.println(idx2 >= 0 ? "Found: " + products[idx2]

: "Product not found.");

}

}

**Analysis**

**Linear Search:**

* **Best Case**: O(1)
* **Average Case**: O(n)
* **Worst Case**: O(n)
* **Space Complexity**: O(1)

**Binary Search:**

* **Best Case**: O(1)
* **Average Case**: O(log n)
* **Worst Case**: O(log n)
* **Space Complexity**: O(1)

**Comparison:**

* **Efficiency**: Binary search is more efficient for large, sorted datasets due to its logarithmic time complexity.
* **Data Requirements**: Binary search requires the data to be sorted, whereas linear search does not.

**2: Financial Forecasting**

**Scenario:**

You are developing a financial forecasting tool that predicts future values based on past data.

**Steps:**

1. **Understand Recursive Algorithms:**
   * Explain the concept of recursion and how it can simplify certain problems.
2. **Setup:**
   * Create a method to calculate the future value using a recursive approach.
3. **Implementation:**
   * Implement a recursive algorithm to predict future values based on past growth rates.
4. **Analysis:**
   * Discuss the time complexity of your recursive algorithm.
   * Explain how to optimize the recursive solution to avoid excessive computation.

Recursion is a programming technique where a function calls itself to solve smaller instances of the same problem. Each recursive call should progress towards a base case, which is a condition that terminates the recursion.

### How Recursion Simplifies Problems

Recursion is particularly effective for problems that can be broken down into similar subproblems. It allows for elegant and concise solutions, especially when the problem has a recursive structure.

Let:

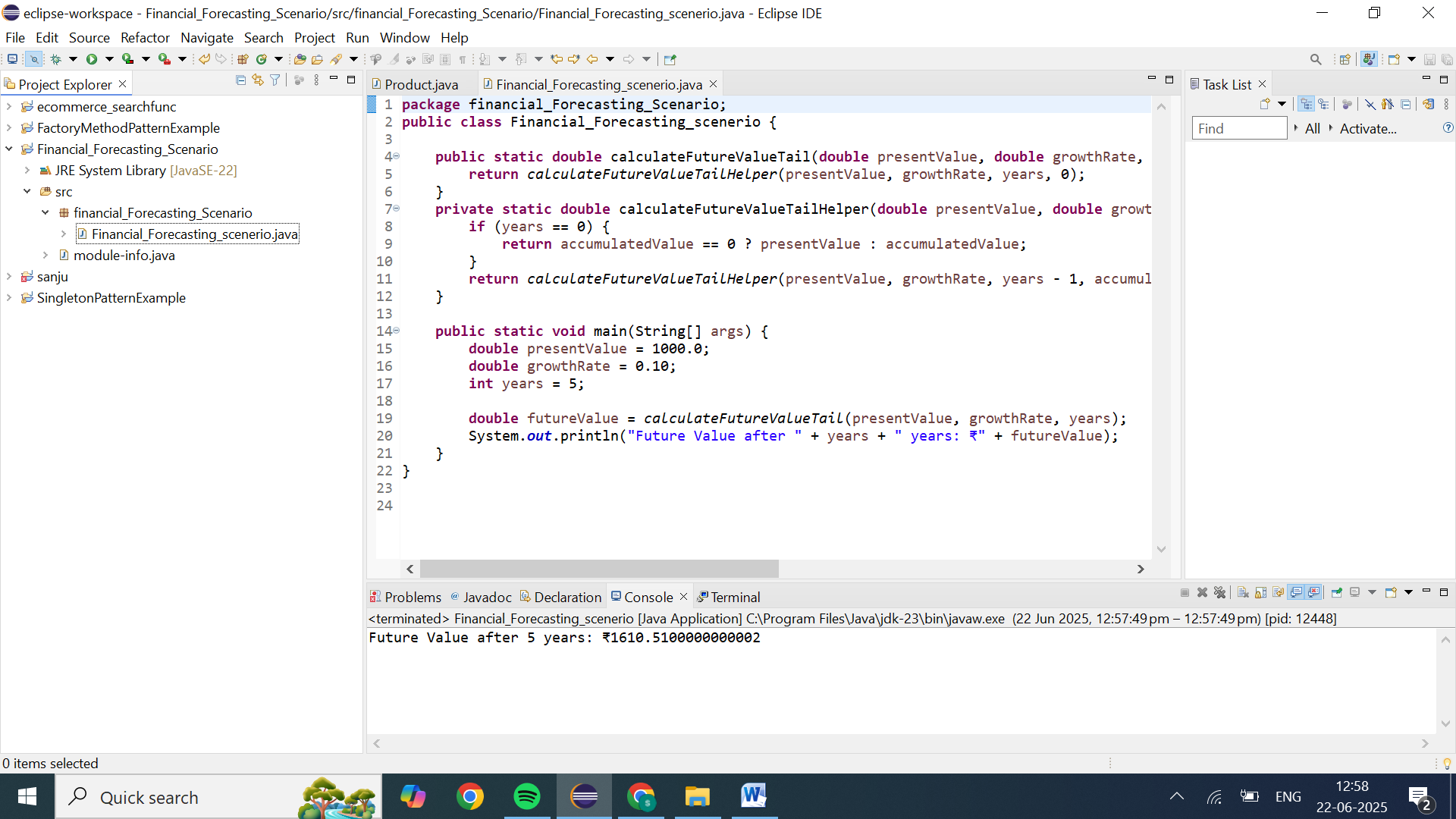
* FV(n) be the future value at time n
* PV be the present value
* r be the growth rate

The recursive formula is:

FV(n)=FV(n−1)×(1+r)FV(n) = FV(n-1) \times (1 + r)FV(n)=FV(n−1)×(1+r)

Where:

* FV(0) = PV (Base case: the present value)



**package** financial\_Forecasting\_Scenario;

**public** **class** Financial\_Forecasting\_scenerio {

**public** **static** **double** calculateFutureValueTail(**double** presentValue, **double** growthRate, **int** years) {

**return** *calculateFutureValueTailHelper*(presentValue, growthRate, years, 0);

}

**private** **static** **double** calculateFutureValueTailHelper(**double** presentValue, **double** growthRate, **int** years, **double** accumulatedValue) {

**if** (years == 0) {

**return** accumulatedValue == 0 ? presentValue : accumulatedValue;

}

**return** *calculateFutureValueTailHelper*(presentValue, growthRate, years - 1, accumulatedValue == 0 ? presentValue \* (1 + growthRate) : accumulatedValue \* (1 + growthRate));

}

**public** **static** **void** main(String[] args) {

**double** presentValue = 1000.0;

**double** growthRate = 0.10;

**int** years = 5;

**double** futureValue = *calculateFutureValueTail*(presentValue, growthRate, years);

System.***out***.println("Future Value after " + years + " years: ₹" + futureValue);

}

}

Recursion provides a clear and straight forward method for financial forecasting by modeling the growth process over time. However, it's essential to consider the potential drawbacks, such as stack overflow and redundant calculations. By employing optimization techniques like memoization, tail recursion, or converting to an iterative approach, you can enhance the efficiency and robustness of your forecasting tool.